ROSETTA SMALLTALK
PROTOTYPE LANGUAGE REFERENCE MANUAL

Rosetta, Inc.
5925 Kirby Suite 215
Houston, Texas 77005

TABLE OF CONTENTS

INTRODUCTION

This manual describes a prototype version of Rosetta Smalltalk and should itself be considered a prototype. It is admittedly incomplete. This manual is not a tutorial. We suggest that you "try it and see" when you have a question about how something works. Our intended audience is those people running our prototype under the CP/M (*) operating system.

Our description of Rosetta Smalltalk is in four parts. First, we will show you how to start up Smalltalk and give you an overview of the Smalltalk world. In the following section we present a short but more formal description of the Rosetta Smalltalk language. In the third section we will show you how to use the class editor for creating and modifying Smalltalk classes. Finally, we have provided in the fourth section an alphabetized "phone directory" of all the pre-defined Rosetta Smalltalk objects with brief descriptions of their behaviors.

----

* CP/M is a trademark of Digital Research

1. THE ROSETTA SMALLTALK WORLD

What follows is a loosely organized introduction to Rosetta Smalltalk. We urge you to read this part of the manual while sitting in front of your computer. Feel free to try things out; we'll suggest a number of examples along the way. If you should do something Smalltalk doesn't like you will get a terse error message, the meaning of which can be found in Appendix A.

Getting started.

The file RS.COM on your distribution disk is the Rosetta Smalltalk prototype. The files with the extension ".WRK" are workspaces containing demonstration applications. After making a backup of the distribution disk, load a disk with RS.COM on it and type RS. When the program has loaded the screen will clear and a rectangular box will appear at the bottom with a copyright message and the characters "?_ " in it. If your terminal supports a mouse you will also see a blinking character in the bottom right corner of the screen. Ignore the mouse for now.

> NOTE:
>
> If you continue to hear the disk reading after you see the box appear on the screen, you probably typed something between the "RS" and the carriage return. This is because Rosetta Smalltalk looks for the name of an initial workspace to load following the "RS" command. If no file with this name is on the disk, an error message will appear. If the file exists but is not a previously saved workspace, you would be well advised to press RESET and reboot CP/M.

The Rosetta Smalltalk console.

Your ASCII keyboard has several keys which designate special functions to Smalltalk; their use will be explained in the following paragraphs. The keys and the character codes they produce will differ on different systems; for this reason we have adopted function-oriented names for them. Appendix B lists the keys selected for the special functions on your terminal.

Typing in a dialog window.

The rectangular box showing at the bottom of your screen is a dialog window. Dialog windows are used for typing in Smalltalk code for immediate evaluation. Your keystrokes and the result of each evaluation will be printed in the window. The "?" you see is a prompt indicating that the dialog window is ready for you to type something. The "_" is a typing cursor.

Dialog windows behave like teletypes. When you're typing on the bottom line of the window and hit the carriage return key, your text will scroll up inside the window. Some other things you will immediately want to know about typing in a dialog window are:

- You can take back the last character you typed by pressing the BACKSPACE key.

- You can erase the entire line on which you are typing by pressing the CLEAR-LINE key.

- You cannot back up past the start of a line. You can, however, throw away everything you've typed so far by pressing the RE-READ key. You will see the letters "DEL" and a new prompt appear.

Try each of these things once or twice.

Word wraparound.

Move the typing cursor two or three characters away from the right edge of the dialog window. Now type a long word like "Smalltalk" and watch what happens. This behavior is commonly referred to as word wraparound. Whenever a word will not wholly fit on the end of a line in a window, that word will be moved down to the next line. For the purposes of wraparound, a "word" is any sequence of non-blank characters. Remember to press RE-READ before going on so that what you've typed doesn't get sent to Smalltalk.

Doing it.

Now type something simple like 3 + 4. A dialog window will not do anything with your keyboard input until you press the DOIT key; this is the key labelled LINEFEED on most keyboards. Until DOIT is pressed, you can edit your input in the manner we described above until you've got it right. You can use the RETURN key to go to a new line any place where a blank would be

acceptable. Press DOIT now; it èchoes as a "!". The result of evaluating what you typed is printed on the following line.

Talking to objects.

Rosetta Smalltalk is an environment for interactive problem solving. This environment is populated by a diverse assortment of active agents which are generically referred to as objects. You've met a few of these already: the numbers 3 and 4, the dialog window you've been typing in, even the demon who's been watching for your keystrokes. Every single thing that you do in Smalltalk gets done by requesting some object to do it for you. Requests to do things are made to objects by sending them messages. For instance, when you added 3 and 4 a while ago, what actually happened was that you sent the message "+ 4" to the number 3. This is a lot different from performing the operation "+" on two numbers.

The Smalltalk world-view may seem rather unusual to you if you are familiar with other programming languages. You can think of each object as being an intelligent creature inside your computer that knows how to perform certain kinds of tasks. The number 3, for instance, is a creature that knows how to do arithmetic. Objects that behave in the same way are grouped into classes. The ability to do arithmetic is a property of all numbers, since they all belong to the class Number.

Let's explore the idea of sending messages to objects further. The dialog window you've been typing in is called disp for historical reasons. To send disp a message, you write "disp" followed by the message you want to send. When you DOIT, the message is sent, disp receives it and carries out your request, and finally sends back a reply of some sort. For example, to move disp up to the top of the screen you can say

```
disp move to 2 2!
```

Try this. Remember to press DOIT. (We will underscore the "!" character in our examples when we want to indicate the DOIT key and not the "!" key).

If you're wondering what disp's reply to you was, the text "<Window>" that was printed in disp is a clue. In a dialog window the reply from each "do-it" gets sent, behind the scenes, the message "print". Objects respond to the "print" message by printing some textual representation of themselves in the object named "disp". In the example above it was disp itself that was asked to print; this is because windows reply to the "move to" message with themselves. Objects like numbers have natural ways of printing themselves, but many objects have no obvious

printable representation. By default, an object will print itself by printing the name of its class in angle brackets. You can supply more helpful printing methods if you so desire.

Since everything is an object, the reply to a message-sending must also be an object. The choice of a reply is arbitrary but is generally whatever is most helpful. Hence, the number 3 will reply to the message "+ 4" with the number 7. Since the reply to a message-sending is an object, you can send this object a message also. Try the following:

```
disp unframe frame!
```

In this bit of Smalltalk, disp first receives a message to unframe itself, and responds by erasing its frame. Since windows reply to this message with themselves, disp immediately becomes the receiver of the next message, and so re-draws its frame. Again the reply is disp itself, which behind the scenes receives the message to print, and hence prints "<Window>".

The mouse.

The blinking character on your screen is the mouse; it is used primarily as a pointing device. You can drive the mouse around by pressing the MOUSE UP, DOWN, LEFT, and RIGHT keys on your keyboard. Try this. You can move the mouse at any time, even while simultaneously running Smalltalk code. Try the following:

```
repeat (disp <- "*")!
```

(Note: the left arrow "<-" is two keys on most ASCII keyboards, but may be produced by one key if your terminal has a left arrow graphic character. Refer to Appendix B). Now drive the mouse around on the screen with the mouse keys. You will continue to see stars print in disp; you'll also notice that the mouse moves somewhat more slowly now that you're doing two things at once. Press the STOP key to regain control.

You can ask the mouse where it is from Smalltalk. The object ml will tell you what screen line the mouse is on; mc will tell you the mouse's column position. A common use of ml and mc is to drag a window with the mouse. Try this:

```
@w <- Window new 5 10 2 2 show!
repeat (w move to ml+1 mc+1)!
```

Now move the mouse and watch how the window w follows it. If you try dragging w across disp you will discover a very nice property of windows: when a window moves or hides, the windows showing behind it are immediately re-displayed. If you have a serial terminal, dragging a window with the mouse may be a bit sluggish. If this is the case, a better way to position a window is to first move the mouse, and then move the window to where the mouse is. We can do this with the following Smalltalk code:

```
repeat (mb. w move to ml+1 mc+1)!
```

The object mb waits for the MOUSE BUTTON key to be pressed and released. The effect of the above code is to wait on the mouse button each time before moving w. In this way you can drive the mouse wherever you want, then press the mouse button, and the window w will leap to where the mouse is sitting.

Sometimes it's useful to ask if the mouse button is pressed, but not wait if it is not. You can do this by sending the "?" message to mb; the reply will be yes if the mouse button is pressed, and no otherwise. For example:

```
repeat (mb? => (w move to ml+1 mc+1)
        disp <- "*")!
```

With this code we ask if the mouse button is pressed; if it is we move w to the mouse, otherwise we print stars in disp. Type this in, play with the mouse keys, and watch what happens.

The meaning of "mb ?" is a bit different for serial terminals. On these terminals, once a key is pressed it is saved until another key is pressed. Hence it is impossible to ask such terminals "Is this key pressed right now?"; one can only ask "Has this key been pressed?". The answer to this latter question is what you will get from "mb ?" if you are using a serially interfaced terminal.

One final way to use the mouse that you should know about is the "has mouse" message answered by windows. A window replies to this message with yes if the mouse is anywhere on top of its frame or text area; it replies no otherwise. Hence, by sending this message to a window, you can ask it if the mouse is touching it. We use this capability extensively in Rosetta Smalltalk to point to windows in order to "wake them up" so that we may interact with them. As a very simple example, try the following:

```
repeat (w has mouse => (w unframe frame))!
```

Type this in and then move the mouse on and away from w. Whenever the mouse comes on top of w it will flash its frame.

2. THE ROSETTA SMALLTALK LANGUAGE

The Rosetta Smalltalk language is based on the notion of objects which send and respond to messages. Literally everything in Smalltalk is an object. An object cannot be operated upon directly, but can only be sent requests to perform actions and return replies. Every object is a member of some class which describes its representation, the messages it can receive, and the methods it uses to answer them. Rosetta Smalltalk is easily extended with new classes of objects and new syntax for messages.

Message sending.

A message is sent by writing the message receiver followed by the message itself. For example, to move the window disp to a different place on the screen you can say

```
disp move to 10 2!
```

In this message-sending the message receiver is disp and "move to 10 2" is the message being sent. We represent the syntax of this message by the message pattern

```
... move to (line) (column)
```

The "..." indicates the message receiver, move and to are literal message tokens, and the parenthesized variables line and column denote evaluated message parameter slots. Any Smalltalk expression can appear where an evaluated parameter is allowed. For example, in the previous section we used the following message to move the window w to where the mouse is.

```
w move to ml+1 mc+1!
```

An object can also receive a single token as an unevaluated message parameter. For example, the object do answers a message of the form

```
... (n) (@code)
```

There are two components to this message; both are parameters. The first parameter n is evaluated, like line and column in the "move to" message. The second parameter code is received by do unevaluated; this is indicated by the "@" symbol preceding the variable name in the message pattern. This parameter is received unevaluated because do will evaluate code itself; in fact, do's role in life is to evaluate the code you send it the specified number of times, as in

```
do 3*4 (disp unframe frame)!
```

When do receives this message, n is 12 and code is the literal list "(disp unframe frame)"; do answers this message by evaluating the code 12 times, causing disp to blink its frame off and on.

The first step Rosetta Smalltalk takes in carrying out a message-sending is to determine what object is to be the message receiver. In effect, the token occupying the receiver position is sent the message ...eval; its reply becomes the message receiver. A "token" may be a Number, String, Atom, or List. If the token is a Number or String its reply is itself. Atoms are used as variable names in Rosetta Smalltalk; an Atom therefore replies to ...eval with the object to which it is bound (i.e. its "value"). Lists respond to ...eval by running themselves as Smalltalk code. Some examples follow.

message	receiver
3 + 4	3
"abc" length	"abc"
a print	value bound to a
(3 + 4) * 5	7

Once the receiver is obtained, Smalltalk begins matching message patterns against the following tokens. Only those patterns belonging to the receiver's class are eligible to be matched. Matching proceeds from left to right, interleaved with evaluation of subexpressions corresponding to parameter slots. Smalltalk matches a specific token in preference to a parameter slot, and always takes the longest possible match. The empty message will be matched if the receiver can answer it and no longer match is found. Once a unique pattern is matched Smalltalk sends the message, setting up a new context for the object to respond in.

Objects answer their messages by running Lists of Smalltalk code called <u>methods</u>. The reply from a message-sending is the result of sending the message ...eval to its method. Rosetta Smalltalk uses periods to separate message-sendings when it is not intended for the reply of one message to become the receiver of the next. Thus the expression

```
do 3*4 (disp unframe. disp frame)!
```

has the same effect as the example above. When several message-sendings are separated by periods in this manner, it is the reply from the last message-sending that becomes the reply from the method as a whole.

Rosetta Smalltalk's form of conditional expression provides a way to escape from the middle of a method with a reply. We used the conditional in some of the examples in the previous section. Its syntax is as follows:

```
(expr1) => (@ alternative1)
(expr2) => (@ alternative2)
          ...
(exprN) => (@ alternativeN)
```

If the result of evaluating expr1 is anything other than the object <u>no</u>, the code alternative1 is evaluated and its reply becomes the reply of the entire method. If the result of evaluating expr1 is <u>no</u>, alternative1 is skipped and expr2 is evaluated, and so on. For example, the following expression replies with the smaller of x and y.

```
x < y => (x) y
```

The following might be a method used by a class Stack to answer the message ...pop:

```
self empty => (error "Stack empty")
@x <- array[top].
@top <- top - 1.
x
```

A conditional may of course be used as a subexpression, as in

```
disp move to (l1 < l2 => (l1) l2)  (c1 < c2 => (c1) c2)
```

The context of a message-sending.

Every Smalltalk object owns some private data that can be directly accessed only by itself. These _instance variables_ are property names common to all instances of a class, for which each instance has particular values. For example, a window object's size is described by two variables: _h_, its height in lines, and _w_ its width in columns. Each window has its own values for these variables and refers to them whenever it is asked to show on the screen. We cannot change these values directly, but a window will do so if asked:

```
disp grow to 10 30!
```

Sending this message has the visible effect of setting _disp_'s size to 10 lines of 30 columns each. To accomplish this, _disp_ has to hide itself, adjust its text buffer to 300 characters, update its _h_ and _w_ values, and show itself again. Because unauthorized access to instance variables is prohibited, the window is able to ensure that its buffer size and visible appearance remain consistent with its height and width.

A method refers directly to an object's private data by mentioning its instance variable names. The method can also mention the special name _self_ to refer to the object receiving the message. Objects often send themselves messages this way. For example, the method by which windows respond to the "grow to" message could be

```
... grow to (newh) (neww) =>

    ( self hide.
      @text <- String new newh*neww.
      @h <- newh.  @w <- neww.
      self show )
```

An object may reveal as much or as little of its representation as it desires by the messages it chooses to answer. It can grant full access to its representation by answering

```
... 's (@code) => (code eval)
```

When this message is sent, code is an unevaluated piece of Smalltalk code, and the object replies with the result of evaluating that code in its private context. If this message is defined for windows, sending

```
disp's h!
```

will reply with the height of disp. This kind of message is helpful when debugging, but must be used with care since the object's assumptions about its private data can be disrupted. For instance,

```
disp's (@h <- h+2)!
```

increases disp's height without making a corresponding adjustment in its text buffer, and will cause an error the next time disp is asked to show.

There are actually three sets of variables in the local context of a message-sending: temporary variables, instance variables, and class variables. All three kinds may be accessed directly by a method. Temporary variables are created when a message is sent and disappear as soon as a reply is made. These variables may be used as scratchpad storage while the method is running. Certain temporaries are initialized with values from the message and thus serve as formal parameters; the variables newh and neww in the "grow to" message are examples of this. Instance variables are names for the data private to each instance of a class, as discussed above. Their values persist between message-sendings as long as the object exists. Class variables play the role often filled by global variables in other languages, but in a more secure and modular way. The shared information held in class variables is accessible only to members of the class and not to the world at large.

When a name is mentioned that is not one of the three kinds of locals, Smalltalk looks for it in the dynamically enclosing context -- that is, the one from which the current message was sent. The search ends in the user's workspace. A common problem with dynamic name scoping is the accidental hiding of global variables when code is run inside a context that happens to use those names for another purpose. Rosetta Smalltalk does not suffer from this problem because all class-related data is accessible from the innermost context, including the class variables that would have been global in some other languages.

Classes.

A class is a description of a kind of object, of which there may be many instances. We group objects into classes so they can share the same representation, message patterns, and methods. By creating new classes the Smalltalk user creates objects modelling his own abstract ideas, and invents his own notation for using them as well.

Classes are a tool for extending a language in a modular way. The representation of an object is ordinarily concealed from outside the object. The only operation that can be performed on an object is to send it a message requesting some action; how that action is carried out is of no concern to the sender and may be changed without affecting existing code. This object-oriented style of programming collects related code into a central place, the class definition. For instance, details of how a class of objects should be printed are grouped with other details about the class rather than in some all-purpose print routine. This makes it easier to find all affected code when a change is made.

A new object is created by sending a ...new message to the desired class. The object should respond to a message beginning with the special token <u>isnew</u> by initializing its instance variables appropriately. One cannot forget to initialize an object because the <u>isnew</u> token is automatically supplied by the system. Apart from this bit of synchronization, <u>isnew</u> messages are no different from other messages. For example, a new window must be told its initial size and location. To create a new window and name it <u>mywindow</u>, we say

```
@mywindow <- Window new 5 30 2 2!
```

This creates a new window which immediately receives the message "isnew 5 30 2 2". The new window initializes its height and width to 5 lines of 30 columns and its screen location to line 2, column 2. Other instance variables are computed from the given information. For example, <u>mywindow</u>'s text buffer is allocated to hold 150 characters.

Every object in Rosetta Smalltalk belongs to a class, and classes are no exception. Every class is an instance of the class named <u>Class</u>; this class has the unique property of being an instance of itself. To create a new class we send the ...new message to <u>Class</u>:

```
@Stack <- Class new!
```

Of course, the new class must be given variable dictionaries, message patterns, and methods for it to be useful. This could be done by sending appropriate messages to Stack, though we would ordinarily use the class editor.

One can also extend or modify the definitions of existing classes. This includes predefined classes of Rosetta Smalltalk as well as those created by the user. As a simple example, suppose we want windows to be able to flash themselves in order to attract our attention. We must define two things: the syntax of the message and the method used to answer it. Our new message syntax will be

```
... flash (n) times
```

The method for flashing will be to erase and redraw the window's frame the requested number of times. We can add this capability to class Window by evaluating

```
Window answer @(flash (n) times)
           by @(do n (self unframe frame))!
```

This is just a message to Window. The "@" tokens indicate that the following parenthesized lists should be taken literally rather than evaluated. After adding the above message to class Window we can say

```
mywindow flash 20 times!
```

and our window will blink its frame off and on 20 times. Note that when a new message is added to a class, all existing instances can immediately respond.

3. THE CLASS EDITOR

The class editor is written almost entirely in Smalltalk. It's not resident, but can be found in the workspace EDITOR.WRK on your distribution disk. To load it, type

```
load "editor"!
```

from Smalltalk, or type

```
A>rs editor
```

from CP/M. The editor is a class variable of class Class. You can edit a class by sending it the message ...edit. This will only work in a workspace that contains the editor; in a virgin workspace without the editor a class will answer the ...edit message by doing nothing.

To show you how the class editor works let's use it to add a method for flashing windows. In a convenient dialog window, type

```
Window edit!
```

You will see several windows appear in the upper left corner of the screen, looking something like Figure 1. The window at the top describes what is being edited. Your initial view of the class is its messages, so "Window messages" appears in this window. The large window is the text window that displays what is being edited; you will initially see a list of message patterns. The window at the right is a menu of editing commands. Selections from this menu are made by pressing single keys corresponding to the first character of a command name. Selecting the "..." command will bring a new menu of additional commands into view. Only those commands currently appearing in the menu can be selected. The bottom window is a dialog window used for typing text to be inserted and for displaying error messages from the editor. If an error message appears in this window, press any key to clear the window and resume editing.

As a general rule, menu choices that begin with an upper case letter are used to modify some part of the class definition. Menu choices that begin with a lower case letter are simply used to change your view, for example to scroll the text up or down, or to view a different part of the class definition.

We use the mouse to point at what we want to edit in the text window. To point at a particular token, place the mouse anywhere on top of the token or in the blank spaces to its left. When the mouse is to the right of the last token on a line it is considered to be pointing at the first token on the next line. The editor's text window acts like a viewport placed on top of a scrolling sheet of paper. Lines clipped outside of this window can be scrolled into view by the menu choices "up" and "down". Try using these commands to scroll the messages up and down.

```
+-----------------------------------------+
|Window messages                          |
+--------------------------------+--------+
| ... isnew (a1) (a2) (a3) (a4)  |up      |
| ... is ?                       |down    |
| ... is (a1)                    |Answer  |
| ... print                      |Change  |
| ... <- (a1)                    |Forget  |
| ... show                       |method  |
| ... hide                       |Quit    |
| ... clear                      |...     |
| ... frame                      |        |
| ... unframe                    |        |
| ... at (a1) (a2)               |        |
| ... move to (a1) (a2)          |        |
| ... grow to (a1) (a2)          |        |
+--------------------------------+        |
|                                |        |
|                                |        |
+--------------------------------+--------+
```

Figure 1. The class editor.

Editing messages.

The three menu choices "Answer", "Forget", and "Change" are specifically for editing the messages. Let's now add our new message for flashing windows. Press Answer; the prompt "Answer?" will appear in the bottom window. At the same time, the menu clears to indicate that no choices from it can be made. Type in the message pattern "flash (n) times" and press DOIT (do not type the "..."; this is just a notational convention). Since you are typing in a dialog window here, all of the conventions you are familiar with such as word wraparound and the CLEAR-LINE key work as you would expect.

When you press DOIT, the menu will reappear. Use the "up" and "down" menu choices to scroll the new message into view if it's not already. Pointing at a message with the mouse and pressing "m" for method allows you to edit the method for the message. Forget will cause the class to forget that message. Change allows you to change the syntax of a message pattern, keeping the same method.

Editing methods.

Point the mouse at your new message and press method. The top window will display "<Window> flash (n) times" to indicate that you are now viewing the method that an instance of class Window uses to answer the "... flash (n) times" message. The text window will of course be empty, since there is no method as yet. You will also notice that the menu has a new set of commands. The meaning of most of these commands should be obvious.

Our method for flashing will be to erase and redraw the window's frame the specified number of times. Press Add; you will get the prompt "Add?" in the dialog window. The text you type will be inserted to the right of the mouse. Type

```
do n ()!
```

The bottom window will clear, the menu will reappear, and the text you just typed will appear in the text window.

The editor knows about the structure of Smalltalk code and uses this knowledge to format the displayed code attractively. The Smalltalk code is always shown neatly indented, with each statement starting on a new line. Whenever the text is altered it is immediately reformatted. Only the top level of the code is displayed; parenthesized subexpressions are simpy shown as "()" or "{}". The "in" command descends into one of these subexpressions to see its top level; the "out" command brings you back out again.

Move the mouse on top of the "()" and press in. Since we haven't added this code yet, the text window will again be empty. Press Add and type

```
self unframe frame!
```

This is the code which windows will do n times in order to flash. Now press "o" to go out to the previous level. The subexpression brackets "()" will now be "{}" to indicate that

there is now some code inside this subexpression.

The commands "Delete", "Move", "Replace", and "Paren" all require you to delimit a piece of text. The left edge of the text is marked by the position of the mouse at the time you select the command. Position the mouse to the right of the last token you wish to delimit and press the mouse button to complete the operation. If this last token is not visible, use the "down" command to scroll it into view.

We suggest you experiment with all of the editing commands until you understand their behaviors. To continue with our example, however, the next thing you should do after adding the flashing method is press message to return to the list of messages. From here you can press Quit to get out of the editor. Now try typing

```
disp flash 10 times!
```

The dialog window you're talking to should blink its frame off and on 10 times.

4. THE PRE-DEFINED ROSETTA SMALLTALK OBJECTS

As you know, everything in Rosetta Smalltalk is an object. There are basically two kinds of objects, classes and instances of classes. Of course, classes are also instances of the class Class, which means that Class is an instance of itself. The first part of our summary of the Rosetta Smalltalk objects describes the pre-defined classes. In the second part we will describe the pre-defined utility objects such as repeat and kb which are unique instances of anonymous classes.

4.1. The pre-defined classes.

Classes are descriptions of objects, of which there may be many instances. The definition of a class includes the messages answered by its instances, its variable dictionaries, its class variables, and its title. The title of any pre-defined class is simply its name; for example the title of the class Atom is "Atom". Our convention is to capitalize class names, but you may name a class anything you like.

Most of the methods used to answer messages of the pre-defined classes are primitive methods. If you ask a class to tell you its method for one of these messages, it will reply with a number. You cannot change the primitive methods or make a class forget a message which uses a primitive method. A few of the pre-defined methods are Smalltalk methods which you can change if you like; where this is the case the method will be given along with the message in the summary that follows.

Atom

Atoms are used as variables and as syntactic message tokens. An atom is used as a variable by binding a value (some object) to it. An atom has a spelling which is represented by a String. Atoms are unique; no two atoms have the same spelling.

... isnew (a1)	Replies the unique atom whose spelling is a1, which must be a String.
... <- (a1)	Binds the object a1 to the receiver in the current context. Reply is a1.
... eval	Replies the object bound to the receiver in the current context.
... chars	Replies with the receiver's spelling.
... print	Prints the receiver's spelling in disp.
... = (a1)	Replies yes if the receiver and a1 are the same atom. Replies no if a1 is a different atom or any other object.

Boolean

A message sending which poses a yes-or-no question will reply with one of the objects yes or no. These "truth values" are instances of the class Boolean.

... isnew	Replies the already existing Boolean no.
... print	Prints yes or no.
... and (a1)	Replies the receiver if a1 is not the object no; replies no otherwise.
... or (a1)	Replies the receiver if a1 is the object no; replies ~~no~~ yes otherwise.

The prefix Boolean function "not" is provided by the object not; see section 4.2.

Class

A class is a description of a kind of object; the class Class is a description of classes. The instance variables of a class are its dictionaries for temporary, instance, and class variables, the list of messages answered by its instances, the class variables, and its title. A class will answer messages to allow you to see or modify the different parts of its description. The ... edit message is sent to a class in order to edit it with the class editor.

... isnew	Replies a new, uninitialized class. The new class's dictionaries and class variables are empty lists and its title is "". The new class has default methods for answering the messages ... isnew, ... is ?, and ... print.
... print	Prints the receiver's title in disp.
... new	Replies a new, uninitialized instance of the class. The new instance will immediately receive a message beginning with the token isnew.
... edit	Invokes the class editor on the receiver. The reply is the edited class. This message has no method in workspaces that do not contain the class editor.
... messages	Replies a list of the messages answered by the class's instances.
... answer (a1) by (a2)	Tells the receiver class to answer the message a1 by the method a2; a1 and a2 are lists. Reply is the receiver.

... forget (a1)	Deletes the message a1 from the messages answered by the class's instances. You cannot forget a message which is answered by a primitive method. Reply is the receiver.
... method for (a1)	Replies the method used in answering the message a1.
... tdict	Replies with the dictionary of temporary variables.
... tdict <- (a1)	Replaces the dictionary of temps by a1. Temps are used both as scratchpad variables and as message parameters; any temps used as message parameters cannot be deleted from the temp dictionary.
... idict	Replies with the dictionary of instance variables.
... idict <- (a1)	Replaces the instance dictionary with a1 and replies with the receiver. If instances of the receiver exist, the dictionary will not be replaced and the reply will be no.
... cdict	Replies with the dictionary of class variables.
... cdict <- (a1)	Replaces the class dictionary with a1. Any new class variables introduced are bound to nil; previously existing class variables retain their values.

... title	Replies the class's title.
... title <- (a1)	Changes the class's title to a1, which must be a String. Reply is the receiver.
... cvar (@a1)	Replies the value bound to the class variable a1. If no such variable is in the class dictionary, no is replied.
... cvar (@a1) <- (a2)	Binds the receiver's class variable a1 to the value a2. Makes an entry for a1 in the class dictionary if one is not there already. Reply is the receiver.

File

A file in our prototype is an object you can use to communicate with a CP/M file on disk. It is not the actual file on disk. The same file object can be used at different times to communicate with any number of different CP/M files. At present there is no way to read a CP/M file directory from Smalltalk.

Files can be read and written either sequentially or randomly. CP/M uses a <CTRL-Z> (ASCII 26) to mark the end of text files. The notion of different file types is not built into class File, however, so when you read the last record of a text file you will get all of the characters following the <CTRL-Z>.

... isnew	Replies with a new file object ready to have a file assigned to it.
... open (a1)	Opens a communication channel between the receiver and the CP/M file named a1. The file name a1 must be a String containing a valid CP/M file name. The reply is no if the file does not exist; otherwise the reply is the message receiver.
... create (a1)	Creates a new file named a1. If a file with this name already exists it is first deleted. Replies the receiver.
... close	Closes the communication channel between the receiver and the CP/M file, writing any buffered text not yet written. The Smalltalk file object can be used to read or write another CP/M file if desired. Be careful to close your files before destroying the file object you are using to communicate with them; you might lose some of the text

you wrote to the file, otherwise.

... at (a1) (a2)

Positions the file at extent a1, byte a2. Extents and offsets within extents are numbered from 0. CP/M extents are 16K bytes large, so the byte offset may be any number between 0 and 16383. Reply is the receiver.

... <- (a1)

Writes the text a1 to the file at its current position. a1 may be a String or a single byte. Notice the similarity between this message and the message used to write text into windows.

... next

Replies with the byte from the file's current position, advancing the position one byte. Replies no if no record has ever been written at the file's current position.

... next for (a1)

Replies with a String of the next a1 bytes of the file from its current position, advancing the position. If fewer than a1 bytes follow, everything up thru the last record is returned. Replies the empty String if no record has ever been written at the file's current position.

A file can be deleted using the delete object described in section 4.2.

List

A list is an object containing a fixed number of positions in which you can put any object. The positions are numbered from 1 to the length of the list. Different lists can have different lengths. Viewed as storage objects, lists are a lot like one-dimensional arrays in other languages. Rosetta Smalltalk also uses lists to represent Smalltalk programs.

... isnew (a1)	Replies a new list of length a1. Each position contains nil.
... print	Smalltalk method: disp <- 40. self length = 0 => (disp <- 41) self each a do (a print. disp <- 32). disp <- 127 <- 41. self
... [(a1)]	Replies the object in the a1th position.
... [(a1)] <- (a2)	Puts the object a2 in the a1th position. Reply is a2.
... [(a1) to (a2)]	Replies a copy of the sublist from position a1 to position a2. If a1 > a2 the reply is the empty list.
... + (a1)	Replies a copy of the list formed by appending the list a1 to the end of the receiver.
... length	Replies the length of the receiver.

... eval	Runs the receiver as Smalltalk code.
... each (@a1) do (@a2)	Iterates over the receiver, temporarily binding <u>a1</u> to each element in turn and running the code <u>a2</u>. Reply is <u>nil</u>.

Number

Numbers are integers in the range -16384 to +16383. Arithmetic on numbers that would yield a result outside this range will reply no instead.

... isnew	Replies 0.
... print	Prints the number in disp. Reply is the receiver.
... chars	Replies the String representation of the receiver.
... + (a1)	Replies the sum of the receiver and a1.
... - (a1)	Replies the difference of the receiver and a1.
... * (a1)	Replies the product of the receiver and a1.
... / (a1)	Replies the integer quotient of the receiver and a1.
... mod (a1)	Replies the integer remainder of the receiver divided by a1.
... < (a1)	Replies yes if the receiver is less than a1; replies no otherwise.
... = (a1)	Replies yes if the receiver is equal to a1; replies no otherwise.

... > (a1)	Replies *yes* if the receiver is greater than *a1*; replies *no* otherwise.
... <= (a1)	Replies *yes* if the receiver is less than or equal to *a1*; replies *no* otherwise.
... <> (a1)	Replies *yes* if the receiver is not equal to *a1*; replies *no* otherwise.
... >= (a1)	Replies *yes* if the receiver is greater than or equal to *a1*; replies *no* otherwise.

String

Strings are used to represent text or are used as byte lists. Strings are similar to lists except that each position of a String can only hold a single byte. We use the words "byte" and "character" interchangeably to mean a number between 0 and 255.

... isnew (a1)	Replies a new String of length a1. The bytes of the new String are uninitialized.
... print	Smalltalk method: `disp <- 34 <- self <- 34. self`
... length	Replies the length of the receiver.
... [(a1)]	Replies the a1th byte.
... [(a1)] <- (a2)	Replaces the a1th byte by a2.
... [(a1) to (a2)]	Replies with a copy of the substring from position a1 to position a2. If a1 > a2 the reply is the empty String.
... [(a1) to (a2)] <- all (a3)	Fills positions a1 to a2 with the byte a3. Reply is a3.
... [(a1) to (a2)] <- (a3)	Replaces the substring from positions a1 to a2 by the String a3 of the same length. Reply is a3.

... < (a1)	Replies yes if the receiver is lexicographically less than a1; replies no otherwise. A formal definition is: "" < s1 for any String s1 of length greater than 0; if s1 and s2 are identical up to position k-1, s1 < s2 if s1[k to s1 length] < s2[k to s2 length].
... = (a1)	Replies yes if the two Strings are identical; replies no otherwise. A formal definition is: "" = ""; s1 = s2 if s1 length = s2 length and s1[i] = s2[i] for i from 1 to s1 length.
... > (a1)	Replies no if self < a1 or self = a1; replies yes otherwise.
... <= (a1)	Replies yes if self < a1 or self = a1; replies no otherwise.
... <> (a1)	Replies yes if self < a1 or self > a1; replies no otherwise.
... >= (a1)	Replies yes if self > a1 or self = a1; replies no otherwise.
... + (a1)	Replies a copy of the String formed by appending a1 to the end of the receiver.
... find first (a1)	Replies the number of the first position where a1 occurs in the receiver, or no if no occurence is found. a1 may be a single byte or a String.

... find first non (a1)

Replies the Number of the first position in the receiver where a character not in _a1_ occurs; replies _no_ if every character in the receievr is in _a1_. _a1_ may be a single byte or a String.

... find [(a1) to (a2)] first (a3)

Like the "... find first" message, but only searches the subrange from position _a1_ to position _a2_.

... find [(a1) to (a2)] first non (a3)

Like the "... find first non" message, but only searches the subrange from position _a1_ to position _a2_.

Window

All screen activity is done through windows. Windows display themselves as rectangular areas on the screen, optionally bordered by a frame. Each window has its own size, screen location, text buffer, cursor, and status flags. Each window may be written into, scrolled, cleared, moved, shrunk or enlarged, and so on independently of the rest of the screen.

Windows may be either hiding or showing. Changes made to a window's appearance while it is hiding have no effect on what you see on the screen. When a window is showing, any changes in its appearance are immediately seen on the screen (unless the screen is frozen; refer to the screen object in section 4.2). Such changes include writing into the text buffer, erasing or adding the frame, and moving or growing the window. When a window moves, any windows showing behind it are immediatley redisplayed.

A window can be as small as 1 line by 1 column or as large as 250 lines by 250 columns. You may find it difficult to create a really large window due to memory limitations. Since a window's frame is optional, a window's screen position is given relative to the upper left corner of its text area. Hence, a window moved to line 1 and column 1 of the screen will show all of its text but the top and left sides of the frame will be clipped off the screen. A window may be placed anywhere. Of course, if your screen is, say, 24 lines by 80 columns, a window placed on line 100 will not be visible even if it is showing.

Text written into a window obeys "word-wraparound" rules. The rules are simple: if a String of text will not fit entirely on the end of a line, it is broken at the rightmost blank. Word-wraparound works the same no matter if you send text to a window in chunks or a character at a time.

Window's answer the ...'s message so you can obtain a window's height, width, screen location, and so on. You should not modify any of a window's instance variables using this message, however, or disaster may ensue.

... isnew (a1) (a2) (a3) (a4)	Initializes a new window to a1 lines and a2 columns in size, placed at line a3 and column a4 on the screen. The new window has a frame but is not showing; this allows windows to be made and written into before they are shown. The text of a new window is initially blank.

... print	Smalltalk method: disp <- "<Window>". self
... <- (a1)	Writes the text a1 into the window at the current position of its cursor. a1 may be a String or a single byte. Reply is the receiver.
... show	Displays the window on the screen. Reply is the receiver.
... hide	Erases the window from the screen. Previously obstructed parts of other windows are brought into view. Reply is the receiver.
... frame	Gives the window a frame. Reply is the receiver.
... unframe	Erases the window's frame. Reply is the receiver.
... clear	Fills the window's text buffer with blanks. Reply is the receiver.
... at (a1) (a2)	Sets the window's write cursor to its own line a1 column a2. No changes are made to the screen appearance; however, the next text written in the window will appear at the new cursor position. Reply is the receiver.
... move to (a1) (a2)	Moves the window to line a1 and column a2 on the screen. Reply is the receiver.

... grow to (a1) (a2)	Changes the window's size to be a1 lines of a2 columns. Any text previously in the window is retained if possible. Reply is the receiver.
... ' s (@a1)	Runs the code a1 in the context of the receiver. Smalltalk method: a1 eval
... has mouse	Replies yes if the mouse touches the window's text or frame; replies no otherwise.

4.2. The pre-defined utility objects.

These objects are unique instances of anonymous classes. Since you cannot get at the class to which one of these objects belongs, you cannot create a new instance of that object. In describing the messages answered by these objects we will write the object in the receiver position of the message patterns instead of the "..." which indicates an arbitrary receiver.

@ (@x)

The @-symbol is used to refer to something literally; or put another way, to prevent something from being evaluated. It's like the QUOTE function in LISP. For example, evaluating @(1 + 2) replies the list (1 + 2), but evaluating (1 + 2) replies 3.

cpm

Returns to CP/M.

cr

Prints a carriage return in _disp_.

delete (f)

Deletes the file named _f_, where _f_ is a String containing a valid CP/M file name.

do (n) (@code)

Evalutes _code_ _n_ times.

done

Will exit the innermost loop in which it occurs with the value _nil_. Loops that can be exited in this way include

do and repeat loops and the loop of the List message ... each (@a1) do (@a2).

done with (x)

Like done but replies with x as the loop value.

eq (a) (b)

Replies yes if a and b are the same object; replies no otherwise.

forget (@v)

Removes the variable(s) v from your workspace; v may be an atom or a list of atoms.

kb

Waits for a key to be pressed and released, then replies with the ASCII code of the key pressed. The MOUSE keys are not seen by kb.

kb ?

If a key is pressed its ASCII code is replied. If no key is pressed, no is replied.

load (f)

Loads a previously saved workspace. See save.

mb

Waits for the mouse button to be pressed and released.

mb ?

Replies yes if the mouse button is pressed; replies no otherwise.

mc

Replies with the column position of the mouse.

mem compact

Exits all execution up to the top level and compacts memory. Use this if you get an "Error 4".

mem ?

Replies with the Number of words (a word is two bytes) of free space remaining in your workspace.

ml

Replies with the line position of the mouse.

not (b)

Replies yes if b is no; replies no otherwise.

read

Replies with a List of tokens read from the keyboard. A token is an instance of one of the classes Atom, Number, String, or List. The lexical rules for reading tokens are as follows:

token class	written as
Atom	A sequence of characters beginning with a letter and followed by zero or more letters or digits; <<, <=, <>, =<, ==, =>, ><, >=, >>;

	any other single character that does not have a special meaning to Smalltalk.
Number	A sequence of one or more digits.
String	A sequence of characters between " marks.
List	A sequence of tokens between left and right parentheses.

Keyboard input is echoed in the window named _disp_. The prompt "?" first appears in _disp_ followed by a typing cursor. You may type your input in free format; carriage returns and spaces between tokens are ignored. The BACKSPACE key will take back the last key typed; typing CLEAR-LINE will throw away the entire line on which you are typing; typing RE-READ will throw away everything you have typed and give you a new prompt. When you've typed everything like you want it, press the DOIT key (linefeed on most keyboards) to tell _read_ you're done.

read in (w)

Like _read_ but echoes what you type in the window _w_.

read line

Similar to _read_ but returns a String of the characters you type. Reading is terminated either by DOIT or a carriage return. If terminated by a carriage return the <CR> character (ASCII 13) is included in the String.

read line in (w)

Like _read_ _line_ but echoes what you type in the window _w_.

read of (ob)

In this form of _read_, _ob_ may be a String, a File, or any object that replies to the message ...next with a character.

The effect is as if the characters were typed in to read from the keyboard, except no echoing is done. Reading is terminated by a character value of 0, by reaching the end of the String when ob is a String, or by a reply of no when ob is a File or some other object that is sent the ...next message.

repeat (@code)

Repeatedly evaluates code until either done receives control or the STOP key is pressed.

save (f)

Saves your workspace in a CP/M file named f; f must be a String. An extension of ".WRK" is assumed if no extension is specified. At present, save and load do not permit disk drive names to be included in the file name; saving and loading is done using CP/M's currently logged in drive. Prior to saving the workspace, execution is returned to the top level and the workspace is compacted. You should use the [o] option when PIPing a workspace.

screen freeze

Freezes the screen's appearance. Any changes made to the screen's appearance will be tanked up and will only become visible when the screen is unfrozen.

screen unfreeze

Unfreezes the screen, making visible any changes tanked up since the screen was last frozen. If sendings of the ...freeze and ...unfreeze messages are nested, only the outermost ...unfreeze will actually update the screen. The screen is automatically unfrozen whenever Smalltalk returns to the top level, as when an error occurs or when the STOP key is pressed.

self

Replies with the receiver of the current message sending.

sp

Prints a space in disp.

vars

Replies with a List of all the user-defined variables in your workspace.

APPENDIX A. ERROR MESSAGES.

0. Implementation error or feature not implemented.

1. Incomplete message.

2. @ not followed by a token.

3. Atom not bound to a value.

4. Can't find enough contiguous free space to allocate an object.

5. => not followed by a yes-part.

6. Receiver does not answer this message.

7. Atom name is not a String in "<Atom> isnew (a1)" message.

8. Length of new List or String is unacceptable.

9. Subscript for List or String is out of range.

10. Message parameter should be a Number but is not.

11. In "<String> [(a1)] <- (a2)" message, a2 is not in 0..255.

12. In "<String> + (a1)" message, a1 is not a String or Number in 0..255.

13. Strings with length not equal to one do not answer "<String> ascii" message.

14. Message parameter belongs to the wrong class.

15. Divide by zero.

16. In "<Window> <- (a1)", a1 should be a String or a Number between 0 and 255.

17. Attempt to set Window cursor outside text area.

18. Not used.

19. Iterations are nested too deeply.

20. <u>done</u> evaluated with no surrounding iteration.

21. "read of (ob)" replied a Number outside 0..255.

22. "read of (ob)" did not reply with a Number or <u>no</u>.

23. <u>disp</u> is not bound to a <Window>, so can't echo keyboard.

24. Smalltalk stack overflow.

25. Not used.

26. In "<String> + (a1)" message, concatenated String is longer than 16,383 bytes.

27. A token is too long to read.

28. Yes-part of a conditional is not a List.

29. In "<String> [(a1) to (a2)] <- (a3)" message, the length of String a3 must be the same as the length of the substring to be replaced.

30. Message pattern syntax is incorrect.

31. Workspace not saved due to lack of space or some other CP/M

file error.

32. Workspace is incompatible with present system.

33. In "load (f)", f does not exist.

34. Disk read error while attempting "load (f)".

35. Can't redefine pre-defined workspace variables.

36. Error in "<File> create".

37. Error in "<File> close".

38. Error in "<File> open".

39. Disk write error (e.g. not enough space on the disk).

40. File name does not have proper CP/M format.

41. File not open.

42. Disk error encountered when trying to change file extents.

43. String for "mem graphic (char) <- (pattern)" is the wrong length for a graphic bit pattern.

APPENDIX B. ROSETTA SMALLTALK AND YOUR TERMINAL

Keyboard.

The following table lists the correspondences between keys of the "Rosetta Smalltalk console" and keys or key combinations on the Heathkit H19 terminal.

Key Name	Key on H19
BACKSPACE	BACKSPACE
CLEAR-LINE	CTRL-L
DOIT	LINE FEED
MOUSE BUTTON	CTRL-B
MOUSE UP	CTRL-W
MOUSE DOWN	CTRL-Z
MOUSE LEFT	CTRL-A
MOUSE RIGHT	CTRL-S
RE-READ	DELETE
RETURN	RETURN
STOP	ESC

The notation CTRL-L means the key labelled "CTRL" held down in conjunction with the key labelled "L".

It would be nice if you could use the cursor arrow keys to drive the mouse around. Unfortunately, on the H19 these keys produce two-character escape sequences which are sent too fast to be recognized by Smalltalk. This difficulty could be overcome with an interrupt-driven keyboard handler, which we may someday provide.

Display.

Thirty-two of the H19's thirty-three graphics characters can be displayed using ASCII control codes 0 thru 31. Control code 0 corresponds to graphic 94, control code 1 corresponds to graphic 95, and so on.

Note that to display the control code ASCII 13 in a window w you cannot say "w <- 13", since 13 is interpreted by windows to mean "go to the next line". Since a window's text buffer is a String object, you can force a window to display the code 13 by storing it directly in the text buffer. For example

```
w's (text[c + (l-1) * w] <- 13).
w show!
```

displays the code 13 at the current position of window w's cursor (l and c). Inside the parens in the first line above, w refers to window w's width, not the window w itself.

APPENDIX C. THE DEMONSTRATION SMALLTALK APPLICATIONS

Your distribution disk should have on it the following demonstration applications:

DIAGRAMS.ASC
EDITOR.ASC
FORMULA.ASC
LITTLED.ASC
MENU.ASC
QUEUE.ASC
TURTLE.ASC
UTILITY.ASC

If you are running Rosetta Smalltalk on an Exidy Sorcerer(*) you will also have these demos:

FONTEDIT.ASC
ROCKET.ASC
SAVEFONT.ASC

Documentation on the demos is forthcoming; we hope the information that follows will be enough to get you off the ground.

ASC and CNV Files.

The files with the extension .ASC, which stands for ASCII, contain source text for the demos. These files are almost, but not quite, suitable for reading into a Rosetta Smalltalk workspace. To convert an .ASC file into something suitable, you must run the program ASCIICNV.COM on it. You will find this program on your distribution disk along with the demos. ASCIICNV does the following:

- strips out all LINEFEED characters (ASCII 10);

- replaces all occurrences of ! with the character code 0.

The version of ASCIICNV supplied with Exidy Sorcerer versions of Rosetta Smalltalk also replaces all occurrences of the two characters "<-" by the single character code 255.

----
* Sorcerer is a trademark of Exidy, Inc.

To convert, say, the file UTILITY.ASC, you would say to CP/M

```
A>asciicnv utility
```

ASCIICNV assumes its input file has the extension .ASC; it produces an output file with the extension .CNV.

Bootstrapping the demos.

We refer to the process of reading a file of source text and converting it to real Smalltalk objects as filing in. Similarly, writing out objects as source text is filing out. To make these tasks easier, we have provided two objects, filein and fileout. The definitions of filein and fileout appear in UTILITY.ASC, along with a few other useful objects. Before you can use filein to read in a demo, you must first read in filein itself. Here's how you do it.

First, run the ASCIICNV program on UTILITY.ASC, as shown above. You will then have a file called UTILITY.CNV. Now run Rosetta Smalltalk. Once in Smalltalk, you can say the following:

```
@f <- File new open "utility.cnv".
repeat (read of f eval)!
```

This code repeatedly reads something from UTILITY.CNV and evaluates it. Each "read of f" reads everything from the file's current position thru the next 0 byte. The infinite repeat loop is terminated when the "done" at the end of the file is evaluated.

You can save the workspace just created so that you don't have to repeat this process every time you want to file in something. You should first close f and forget it, e.g.

```
f close!
forget f!
save "utility"!
```

If you now look at your workspace variables, you should see that, aside from disp, you also have the objects to, for, indisp, filein, and fileout. In the future all you have to do to load up these objects is say (from CP/M)

```
A>rs utility
```

Filing in.

Assuming you have filein in your workspace, and a .CNV version of a demo, say turtles, you can file in that demo by saying

```
        filein: "turtle"!
```

When you use filein, a window will appear in the upper right corner of the screen displaying how many words of memory are free. This display is updated after each definition is read and evaluated; this lets you know that filing in is still in progress.

Filing in a large demo may fail due to memory fragmentation. A way to overcome this is to put "mem compact" messages every so often in the file; the files EDITOR.ASC, FORMULA.ASC, and FONTEDIT.ASC all do this. When "mem compact" is read and evaluated, all of the free space will be compacted into a single large chunk. Unfortunately, memory compaction requires that first all execution be aborted to the "top level"; hence, after a compaction has occurred you will have to manually restart the filing in process. If you use filein, all you have to do to resume filing in is say

```
        go!
```

You will know that filing in has not completed as long as the "Free Words" window appears at the top of your screen and the object go is still in your workspace. You may find it instructive to look at the definition of filein; note how it creates and destroys go.

Filing out.

The right way to file objects out is to have every object answer the message ...fileout, just like every object now answers the message ...print. Filing out is not something we planned for in the prototype version of Rosetta Smalltalk, so the right way of doing file out is not implemented. What can be done is to use the object fileout, which is defined in UTILITY.ASC.

fileout can be used to file out classes and objects created with to. All parts of a class are filed out except the class variables. There are two ways fileout can be used. One is to file out a single class to a file and close that file. For

example, to file out the class Turtle you can say

```
fileout: Turtle as "turtle.asc"!
```

If you want to file out several things to the same file, you can create the file yourself and use the other fileout message. In-between filing out classes you can file out comments, do simple formatting, and file out arbitrary Smalltalk expressions by sending the appropriate text to the file. For example, here's how you might file out classes Box and Arrow to the file DIAGRAMS.ASC:

```
@QUOTE <- 34.
@CRLF  <- ("" + 13) + 10.
@CTRLZ <- 26.

@f <- File new create "diagrams.asc".
f <- QUOTE <- "DIAGRAMS.ASC" <- QUOTE <- "!".
f <- CRLF <- CRLF.

fileout: Arrow to f.

f <- CRLF <- CRLF.

fileout: Box to f.

f <- CRLF <- CRLF.
f <- "done!".
f <- CRLF <- CTRLZ.
f close!
```

Note that you must run ASCIICNV on a file written with fileout before you can file it back in.

A brief description of each demo.

DIAGRAMS.

The classes Box and Arrow let you make simple data-structure diagrams. A box has a number of fields, each of which is represented by a window. A label may be given to each field. An arrow can be drawn from the field of one box to another box. Arrows can only go to the right and down. Try the following examples.

```
@box1 <- Box new 5 10 2 2 show!
@box2 <- Box new 3 10 10 30 show!
```

```
box1[1] <- "field1"!
box1[2] <- "field2"!

@arrow1 <- Arrow new from [box1 2] to box2!

screen freeze.
box1 move by 3 20.
box2 move by 3 20.
arrow1 move by 3 20.
screen unfreeze!
```

EDITOR.

The file EDITOR.ASC contains source text for the class editor described in section 3 of this manual. The editor is a large Smalltalk program; you won't have much luck using it in anything smaller than a 52K CP/M version of the prototype. If you have less than 52K, you can run the stripped down version of the editor found in LITTLED.ASC.

FONTEDIT.

The font editor runs only on Exidy Sorcerer versions of Rosetta Smalltalk. This application of Smalltalk demonstrates some interesting ways of using windows and the mouse. To start up the font editor you say

```
edfont!
```

A menu and a small blank window will appear in the upper right corner of the screen. Your keystrokes are interpreted in one of two ways. Usually, a keystroke is a pick from the menu. However, when the mouse touches the window below the menu, your keystrokes are echoed in that window and nothing else happens. This allows you to see what the characters you are defining will look like in their actual size. The font editor is a bit slow; you may have to hold down a key for a couple of seconds before it's seen.

The rest of the screen is used for editing characters. Move the mouse to a free area and press "n" for "new". A prompt will appear in the menu asking you to press the key of the character you want to edit. Press the desired key, say "a". An 8 x 8 window, referred to as a "cell", will appear where the mouse is, and will be filled in with a blown up version of the letter "a".

You cannot modify the graphic appearance of the letter "a" since this definition is in the Sorcerer's ROM. However, the menu choice "code" lets you assign the graphic definition of a cell to another character code. Pick "code" and press "GRAPHIC-SHIFT-a" in response to the prompt. By doing this you have copied the appearance of the letter "a" into the graphic definition for the user-definable character "GRAPHIC-SHIFT-a". To see that this is so, move the mouse over into the typing window and type both "a" and "GRAPHIC-SHIFT-a"; the two characters will look the same. The menu choice "?code" lets you find out what character code is assigned to a cell.

Characters are edited using the mouse. Move the mouse inside the cell for "GRAPHIC-SHIFT-a". Pressing the mouse button complements the pixel underneath the mouse. Notice that the appearance of the real character in the typing window immediately changes when you modify its enlarged appearance in the editing cell.

Defining multiple character fonts is aided by the fact that new graphic cells are automatically lined up with adjacent cells. Move the mouse near the cell for "GRAPHIC-SHIFT-a" and press "new". When you press the key to be edited the new cell will appear lined up with the other cell.

The menu choices "putdisk" and "getdisk" let you save and load font definitions on disk. The file name you type in response to the prompt will have the extension .FNT appended to it. The objects <u>savefont</u> and <u>loadfont</u> in the font editor workspace let you save and load fonts quickly, without having to actually run the font editor.

The meaning of the other menu choices should be obvious; if not, just try them out and see what happens.

<u>FORMULA.</u>

The formula demo is a toy symbolic formula manipulation system. We don't claim you could do any serious formula manipulation with it, but it gives you an idea of some of the things you <u>might</u> do. There are two new objects used in the world of formulas. One is the class <u>Formula</u>, which knows about symbolic operations on formulas. The other is the object <u>#</u>, which is used to turn a number or an atom into a formula. Try the following:

```
3 is ?!

#3 is ?!
```

This example makes it clear that 3 is a number, but #3 is a formula. Now try

```
3 + 4!

#3 + #4!
```

You'll see that formula arithmetic is done symbolically. Now say

```
@f <- #1 + #x.  @g <- #y.  f/g!
```

Formulas print themselves just like you would have to type them in, but formulas also know how to display themselves in two-dimensional notation. Try this:

```
@w <- Window new 15 40 2 2 show!
(f/g) show in w!
```

This is cute:

```
@f <- #x.
do 3 (@f <- #1 + #1/f).
f show in w!
```

A few more things formulas know how to do include differentiation, substitution, and simple-minded simplifications. The object dx is an example of a very simple tool built on top of formulas. It allows you to type in a formula, see it displayed in 2-d notation in one window, and see its (somewhat simplified) derivative with respect to x in another window. To get out of dx, type "done" instead of a formula.

LITTLED.

LITTLED.ASC contains a stripped down version of the class editor. This version allows you to edit the messages and methods of a class, but not any of the other class parts. LITTLED is small enough to run in a 48K system.

MENU.

The class Menu lets you build menus like those used by the class editor and the font editor. A menu displays its choices inside a window. Try the following simple example.

```
@m <- Menu new in Window new 10 8 2 2 show!

m when "show"     do @(w show)
  when "hide"     do @(w hide)
  when "frame"    do @(w frame)
  when "unframe"  do @(w unframe)
  when "move"     do @(w move to ml+1 mc+1)
  when "Quit"     do @(done)!

@w <- Window new 10 20 10 40 show!

repeat (m pick)!
```

Now try pressing letters from the set "shfumQ" and watch how the window w obeys your menu picks.

QUEUE.

This demo builds on the diagrams demo. Class Queue is defined to perform the usual functions of a queue, namely put things in at one end and take them off the other end. This version of the queue data structure is animated, however. Move the mouse up to the top left corner of the screen and say

```
@q <- Queue new!
```

A "queue header" for an empty queue will appear where the mouse is. Now try the following:

```
q put 1 put 2 put 3!

q take!
```

ROCKET.

This is a one minute demo showing you how you can use the font editor to define animated figures on the Exidy Sorcerer. The font ROCKET.FNT should be on your distribution disk; it contains a font definition for a little rocket with flames coming out from its tail. Use loadfont or the menu choice "getdisk" in the font editor to load in ROCKET.FNT. Then file in the rocket demo and say

```
launch!
```

Hit the STOP key when you've seen enough.

SAVEFONT.

The objects savefont and loadfont can be used to save and load fonts built with the font editor. Here's how to use loadfont to load in ROCKET.FNT:

```
loadfont: "rocket"!
```

or you can just say

```
loadfont!
```

and you will be prompted for the file name of the font. The use of savefont is similar.

TURTLE.

A turtle is a creature that can crawl around in a window leaving a trail wherever it goes. Try the following classic turtle example:

```
@Ted <- Turtle new in Window new 20 40 2 2 show!
do 4 (Ted go 8 turn 2)!
```

A turtle can go in eight directions; north is direction 1, northeast is direction 2, and so on. Telling a turtle to "turn 2" means it will turn clockwise 90 degrees. A turtle's ink is a character which it leaves behind as it moves; you can change the ink by saying

```
Ted ink "@"!
```

The definition of class Turtle is very short and very informative; it's worth studying in some detail.

UTILITY.

UTILITY.ASC contains definitions for the objects to, for, indisp, filein, and fileout. We've already seen how to use the latter two. The object to is useful for creating "verbs" or "procedures". Try the following:

```
to (flash (w))  (do 10 (w unframe frame))!
flash disp!
```

The above message to to creates an object named flash who is the only instance of an anonymous class. You can ask flash for its class by sending it the message ...class, or the usual message ...is ?.

The object for, which is defined in UTILITY.ASC by using to, implements a for-loop control structure. For example:

```
for k <- 1 to 10 do (k print. sp)!
```

Unfortunately, there is no simple way to dynamically create a "local variable" in our prototype, hence the control variable you use in a for-loop may conflict with your other variables if you're not careful.

The object indisp lets you temporarily name an object "disp" in order to print in it. For example:

```
@w <- Window new 10 20 2 2 show!
indisp w (vars print.)!
```

Try this more complicated example:

```
to dialog (indisp Window new 5 40 ml+1 mc+1 show
                    (repeat (read eval print. cr)))!
dialog!
```

This code uses to to create the object dialog which will start up a dialog loop in a new window positioned at the mouse. The dialog loop is identical to the one you normally run in. To get back to the dialog window you came from, simply say "done". The dialog object we just created could come in handy when you're debugging. For example, saying

```
w's dialog!
```

lets you do dialog in the private context of the window w, so you can examine w's instance variables simply by mentioning their names (we don't recommend you do this with windows, but feel free with objects of your own making).

Another use of indisp is to write to a file. If f is a file,

```
indisp f (vars print.)!
```

will print the list of your workspace variables to the file. Notice how fileout uses this feature.

INDEX